Modelling distributed network attacks with constraints

Pedro Salgueiro* and Salvador Abreu

CENTRIA and Departamento de Informática,
Universidade de Évora,
Rua Romão Ramalho, No. 59, 7000-671 Évora, Portugal
E-mail: pds@di.uevora.pt
E-mail: spa@di.uevora.pt
*Corresponding author

Abstract: NeMODe is a declarative system for computer network intrusion detection, providing a declarative domain specific language for describing network intrusion signatures which can span several network packets, by stating constraints over network packets, describing relations between several packets in a declarative and expressive way. It provides several back-end detection mechanisms, all based on a constraint programming framework, to perform the detection of the desired signatures.

In this work, we demonstrate how to model and perform the detection of distributed network attacks using each of the detection mechanisms provided by NeMODe, based in Gecode, adaptive search and MiniSat to perform the detection of the specific intrusions. We also use the *sliding* network traffic window version of the adaptive search back-end detection mechanism to simulate live network traffic and evaluate the performance of the system in conditions near to real life networks.

Keywords: constraint programming; propagation-based solvers; constraint-based local search; CBLS; Boolean satisfiability problems; intrusion detection systems; IDSs; domain specific languages.

Reference to this paper should be made as follows: Salgueiro, P. and Abreu, S. (2013) 'Modelling distributed network attacks with constraints', *Int. J. Bio-Inspired Computation*, Vol. 5, No. 4, pp.210–225.

Biographical notes: Pedro Salgueiro holds a PhD in Computer Science in 2012 by Universidade de Évora, Portugal. Currently, he is a Software Developer in the field of natural language processing. His research interests include constraint programming, network intrusion detection, domain specific languages and natural language processing.

Salvador Abreu is a Professor of Computer Science at Universidade de Évora. He holds a PhD in Artificial Intelligence by Universidade Nova de Lisboa and has authored over 50 articles in international journals and conferences. His research interests include language models for parallel computation, high performance computing, declarative programming, constraint and logic programming as well as applications thereof.

This paper is a revised and expanded version of a paper entitled 'Modeling distributed network attacks with constraints' presented at the Intelligent Distributed Computing V, Delft, The Netherlands, 5–7 October 2011.

1 Introduction

Maintaining the security of computer networks is a crucial task and plays an important role in keeping the users data safe and the network a safe place to work. Such task can be accomplished by network intrusion detection system (IDS) e.g., Snort (Roesch, 1999).

Distributed network attacks are very popular among hacker communities, since they are driven from several places and easily elude the detection mechanisms, since these attacks originate from several places at once, being difficult to identify the network traffic as an attack.

There are certain aspects that should be verified in order to maintain the security of the user's data, as well as the quality and integrity of the services provided by a computer network. Being able to describe these aspects, together with a verification that they are met can be considered as an network intrusion detection task. Describing those conditions, in terms of properties which must be verified in the network traffic, also describe the desired or unwanted state of the network, which can be induced by a system intrusion or another form of malicious access.

While using a declarative programming approach, such as constraint programming (CP) (Rossi et al., 2006) or constraint-based local search (CBLS) programming (Van Hentenryck and Michel, 2005), we can de scribe those condition, in an easier, natural and expressive way.

NeMODe IDS (Salgueiro et al., 2011) is a declarative system that provides a domain specific language enabling an easy and very descriptive way to describe the network intrusion signatures by following the CP methodologies. Besides using CP paradigm to describe the desired network situation, it also in relies CP to perform the detection of such intrusions, providing several back-end detection mechanisms based on CP, such as propagation-based solvers, using Gecode (GC); and CBLS, using adaptive search (AS).

NeMODe is able to analyse network traffic logs, but also provides a mechanism which simulate real-time network traffic monitoring, through the use of a sliding network traffic window, over a large traffic log. This allows to access the performance of the system on a live network link.

This paper is organised as follows. Section 1 introduces the work and makes a brief description of network IDSs, CP and some of its *flavours*. Section 2 describes the main characteristics of NeMODe, together with a brief description of the language used to model the problems and its architecture. Section 3 describes how to model intrusion detection problems using propagation-based solvers, Section 4 describes how to model network intrusion using CBLS and Section 5 describes how to use model a network situation as a SAT problem. Section 6 introduces the sliding network traffic window used in NeMODe, allowing a real-time intrusion detection. Section 7 presents some examples, demonstrating how to model them in NeMODe. It also shows how they can be described in Snort, so we can evaluate the NeMODe against other systems. Section 8 presents the experimental results of both NeMODe and Snort, Section 9 evaluates NeMODe, and Section 10 presents the conclusions and the future work.

Throughout this paper, we mention some TCP/IP and UDP/IP technical terms, such as *packet flags*, *URG*, *ACK*, *PSH*, *RST*, *SYN*, *FIN*, *acknowledgment*, *source port*, *destination port*, *source address*, *destination address*, *payload*, described in Comer (2006).

1.1 Intrusion detection systems

Network IDSs are very important and one of the first lines of defence against network attacks or other types of malicious access, which constantly monitors the network traffic looking for anomalies or undesirable communications in order to keep the network a safe place.

There are several methods to perform network intrusion detection, but, among them two of them are more used (Zhang and Lee, 2000):

1 based on the network intrusion signatures

2 based on anomaly detection.

On network IDSs based on signatures, the network attacks are described using their signatures, particular properties of network packets used to achieve the desired intrusion or attack, which are then looked in the network traffic. IDSs based on anomaly detection, tries to understand the *normal* behaviour of the systems by modelling its behaviour using statistical methods and/or data mining approaches. The network behaviour is then monitored, and if considered anomalous according the network model, the network is probably under some kind of attack. NeMODe uses an approach based on signatures.

Snort (Roesch, 1999) is a widely used network IDS, primarily designed to detect signatures that can be identified in a single network packet, using efficient pattern-matching techniques to detect the desired intrusion signature.

Although Snort provides some basic mechanisms which allow the writing of rules that spread over several network packets, such as the `Stream4` or `Flow` preprocessors, they do so in a very limited and counterintuitive way, not allowing the description of more complex relations between packets, such as the temporal distance between two packets.

Most of the work in the area of IDSs consists in the development of faster detection methods (Arun, 2009), but there is also some work focused on how the network signatures are described and detected, such as in the work (Kumar and Spafford, 1995), where the authors present a declarative approach to specify intrusion signatures which are represented as a specialised graph, allowing the description of signatures that spread across several network packets.

1.2 Constraint programming

CP is a declarative programming paradigm consisting in the formulation of a solution to a problem specified as a *constraint satisfaction problem* (CSP) (Rossi et al., 2006), in which a number of variables are introduced, with well-specified domains and which describe the state of the system. A set of relations, called *constraints*, is then imposed on the variables which make up the problem. These constraints are understood to have to hold true for a particular set of bindings for the variables, resulting in a *solution* to the CSP.

There are several types of constraint solvers, in this work we use:

1 propagation-based solvers

2 CBLS.

1.3 Propagation-based solvers

Problems in propagation-based (Rossi et al., 2006) solvers are described by stating constraints over each variable that composes the problem. These constraints states what values are allowed to be assigned to each variable. Then, the constraint solver will propagate all the constraints and reduce the domain of each network variable in order to satisfy all the constraints and instantiate the variables that

compose the problem with valid results, thus reaching a solution to the initial problem.

GC (Schulte and Stuckey, 2004) is a constraint solver library based on propagation, implemented in C++ and designed to be interfaced with other systems or programming languages.

1.4 Boolean satisfiability problems (SAT)

A SAT (Biere and Press, 2009) problem consists on determining if there is a valid assignment to all variables of a Boolean function, so that such Boolean function is satisfiable, or determining that there is no valid assignment that can make such Boolean function `True`, implying that the Boolean function is `False`.

In order to solve a SAT problem, there is the need to make a description of the problem as a Boolean function composed by Boolean variables which can only take `True` or `False` values. Usually this function is specified in the conjunctive normal form (CNF) (Biere and Press, 2009), a conjunction of clauses, where each clause is a disjunction of literals, and each literal is a Boolean variable or its negation.

There are a number of SAT solvers which participate in several SAT competitions to evaluate their performance. MiniSat (Sörensson and Een, 2005) is widely used SAT solver, awarded in such those competitions (Een and Sörensson, 2006).

MiniSat is implemented in a way to be a small, complete and efficient SAT solver. One of the major concerns of MiniSat authors was to provide a tool that can easily be adapted to the needs of the users and which could be easily interfaced with other tools, making it an adequate tool to use as a back-end detection mechanism to NeMODe.

1.5 Constraint-based local search

CBLS (Van Hentenryck and Michel, 2005) is a fundamental approach to solve combinatorial problems such as CSPs. Although not a complete algorithm and unable to provide a complete or optimal solution, CBLS is a method that can solve very large problems. Usually, this approach initiates with an initial, candidate solution to the problem which is then iteratively improved though small modifications until some criteria is satisfied. The modifications to the candidate solution are usually driven by heuristics that guide the solver to a solution.

AS (Codognet and Diaz, 2001) is a CBLS (Van Hentenryck and Michel, 2005) algorithm, taking into account the structure of the problem and using variable-based information to design general heuristics which help solve the problem. The iterative repairs to the candidate solution in AS are based on variable and constraint error information which seeks to reduce errors on the variables used to model the problem.

2 Intrusion detection with constraints

Detecting network intrusions with constraints consists on identifying a set of network packets in the network traffic, which identifies and makes proof of the desired network signature attack. The identification process is achieved by matching the intrusion signature described through the use of constraints stated over a set of network packet variables, describing relations between several network packets.

In order to use the CP mechanism to perform network intrusion detection, there is the need to model the desired signature as a CSP. A CSP which models a network situation is composed by a set of variables, V, representing the network packets involved in the description of the network situation; the domain of the network packet variables, D; and a set of constraints, C, which relates the variables in order to describe the network situation. We call such a CSP a network CSP. On a network CSP, each network packet variable is a tuple of integer variables, 19 variables for TCP/IP packets and 12 variables for UDP packets, representing the significant fields of a network packet necessary to model the intrusion signatures used in our experiments. For both TCP/IP and UDP packet representation, we only consider the 'interesting' fields, from an IDS point-of-view.

The domain of the network packet variables, D, are the values actually seen on the network traffic window, which is a set of tuples of 19 integer values (for the TCP variables) and 12 integer values (for the UDP variables), each tuple representing a network packet actually observed on the traffic window and each integer value represents each field relevant to intrusion detection. The packets payload is stored separately in an array containing the payload of all packets seen on the traffic window. The correspondence between the packet and its payload is achieved by matching the packet number, i, which is the first variable in the tuple representing the packets and the i^{th} position of the array containing the payloads.

Listing 1 shows a representation of such CSP, where P represents the set of network packet variables, where $P_{n,z}$, is each of the individual integer variables of the network packet variable, in a total of z fields for each network of the n variables, with $z = 19$ for TCP packets and $z = 12$ for UDP packets. D is the network traffic window, where $D_i = (V_{i,1}, \ldots, V_{i,z}) \in D$ is one of the real network packets on the network traffic window, which is part of the domain of the packet variables P. *Data* is the payloads of the network packets present in the network window, where $Data_i$ is the payload of the packet $P_i = (V_{i,1}, \ldots, V_{i,z}) \in D$.

The associated domains of the network packet variables is represented by $\forall P_i \in P \Rightarrow P_i \in D$, forcing all variables belonging to P to obtain values from the set of packets in the network window D.

A solution to a network CSP, if it exists, is an assignment of network packet values, $D_i = (V_{i,1}, \ldots, V_{i,z}) \in D$, to each packet variable, $P_i = (P_{j,1}, \ldots, P_{j,z}) \in P$, that models the desired situation, thus identifying the network packets that identify the intrusion being detected.

Listing 1 Representation of a network CSP

$P = \{(P_{1,1}, \ldots, P_{1,z}), \ldots, (P_{n,1}, \ldots, P_{n,z})\}$
$D = \{(V_{1,1}, \ldots, V_{1,z}), \ldots, (V_{x,1}, \ldots, V_{x,z})\}$
$Data = \{Data_1, \ldots, Data_x\}$
$\forall P_i \in P \Rightarrow P_i \in D$

2.1 A DSL to describe network signatures

The NeMODe IDS is a declarative system that provides a domain specific language, following the CP methodologies, enabling an easy and very descriptive way to describe the intrusion signatures that spread across several network packets by allowing to state constraints over network entities and express relations across several packets.

The key characteristic of this DSL is to ease the way how network attack signatures are described using CP, hiding from the user all the constraint programming aspects and complexity of modelling network signatures as a CSP, but still using the methodologies of CP to describe the problem at a much higher level, describing how the network entities should relate among each other and what properties they should verify.

Maintaining the declaritivity and expressiveness of CP allows an easy and intuitive way of describing the network attack signatures by describing the properties that must or must not be seen on the individual network packets, as well as the relationships that should or should not exist between each of the network packets.

The DSL is a front-end to several back-ends, one to each intrusion detection mechanism, including propagation-based systems, such as GC and CBLS, such as AS. This allows to generate several recognisers based on different constraint solver methods, from a single description. With several recognisers, it is possible to run each of them in parallel, allowing to select the first produced solution, as the behaviour of each solver depends on the problem being solved. This DSL is further described in Salgueiro et al., 2011; Salgueiro and Abreu, 2010).

In Section 7, we present the description of some network attacks using NeMODe.

2.2 Architecture

NeMODe is composed by a compiler, which reads a NeMODe programme and parses it into a semantic model. Then, based on that semantic model, it is generated code for each of the available back-ends in system.

After all recognisers have been generated, each generated back-end receives as input the network traffic and produces a valid solution, if the intrusion described as a NeMODe programme exists on the network traffic that was given as input to each back-end detection mechanism.

All back-ends available in the system work in parallel, each one producing a solution to the problem. In a final step, the best solution produced is selected, which is simply the first solution to be produced.

Figure 1 represents the architecture of the system and how the data flows between each component.

3 Modelling with propagation-based solvers

Propagation-based systems relies on functions which reduce variable domains, which in turn reduces the search space, until no more violations to the constraints used to model the problem are found, and all variables are reduced to a single-valued domain, thus reaching a solution to the problem, if one exists.

In order to model a problem in propagation-based solvers, we need to model the problem as a CSP, and to do so, three things need to be defined:

1 the variables of the problem

2 the domain of the variables

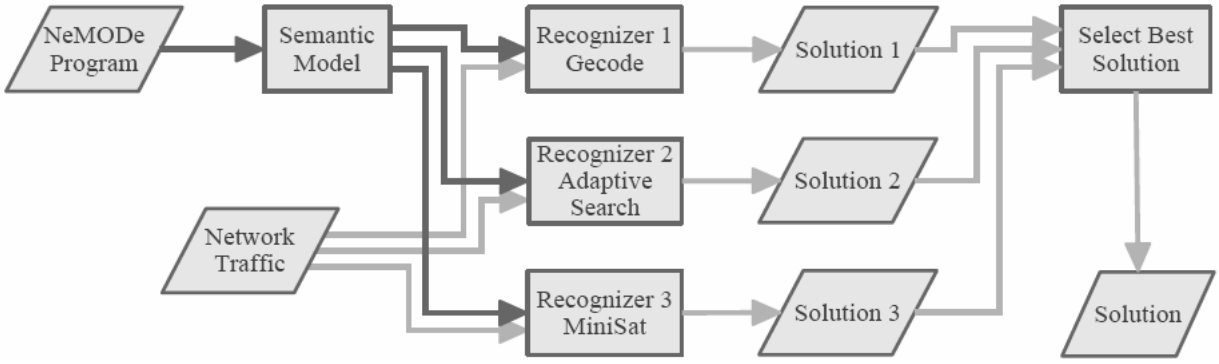
3 the constraints which describe the problem.

In the case of network intrusion detection problems, the same modelling will be used independently of the solver being used.

3.1 Modelling in GC

Modelling network intrusion detection problems in GC is basically done by asserting relations between the variables of the problem in order to describe the desired network intrusion signatures.

Figure 1 NeMODe system architecture



Three main steps need to be done to model a network intrusion detection in GC:

1 model the variables

2 specify the domains of each variable

3 specify the constraints in order to model the problem.

3.1.1 Variable representation

In GC, the primary type of the variables is `Integer`. There are helpers which ease the description of the problem in terms of variables, such as the type `Integer Array`, an array of variables of type `Integer`. This is a perfect data type to represent the network packet variables in a CSP, since it can easily be represented as a tuple of `Integer` variables, as provided by the `Integer Array`.

3.1.2 Constraint specification

The most important part in describing a network intrusion detection problem in GC is the specification of the constraints. Stating the constraint over the network variables is what models and describes a specific attack, which exists in the present network traffic *iff* the constraint problem has a solution.

The constraints are stated over one or more network packet variables, and are responsible to specify and ensure which properties are verified by each packet and which relations should hold between multiple packets, as used to model the problem.

3.1.3 Modelling a problem

The problem modelling is achieved through the use of the functions which implements the constraints. First, the functions which encode and define the variables are applied, followed by the functions which implement the constraints and actually model the problem.

4 Modelling with CBLS

When using CBLS to perform intrusion detection, two major things need to be defined:

1 the number of network packets that needs to be found in order to identify the network situation being sought

2 the constraints that define the problem.

In CBLS, the constraints are built in order to drive an heuristic search, by providing the number of violations of each variable used by the constraint. This is a critical aspect on CBLS since these values will act as heuristics which guide the search algorithm to reach a solution to the problem, so, a critical step in modelling intrusion detection with CBLS is to find good heuristics with which to solve the problem.

In order to solve a problem, CBLS starts by creating a first tentative solution by assign values, usually randomly chosen, to the variables of the problem. It then performs small changes to that solution in order to converge on a final solution, using heuristics to decide which changes will be made. This step is repeated until an objective function is reached, reaching a valid solution.

The initial tentative solution is very important in the way the solution is reached, so, choosing an initial solution that best suits intrusion detection problems is also very important.

4.1 Modelling with AS

AS requires problems that can be stated as a permutation. Because of this, when modelling a problem with N variables, the domain of each variable will be $D = I, \ldots, I + N - 1$, where I is the lower value that each variable can take. A solution to such problem will be a permutation of D. This characteristic of AS poses some limitations in the way that a network intrusion can be modelled.

Modelling a network intrusion as a CSP relies on defining a set of network packets variables, P, which describes the network situation itself, and the network traffic window, D, which contains all the network packets seen on a network traffic window.

P is the network packet set being looked for in D, which is much larger than P. This situation is incompatible with AS, as the domain set is larger than the variable set, and, a solution to such a CSP will be a subset of all network packets available on a network traffic window.

To work around this problem, the network situation is modelled using as many variables as the number of network packets in the network traffic window, but, most of the variables are ignored, so that only the number of variables that are being used to model the problem are used during the solving process of AS. So, if the network situation being modelled is composed by a set of N network packet variables and the network traffic window is composed of M network packets, the problem will be modelled using M network variables, still, only the first N variables will be used to reach a solution, ignoring the last $M - N$ variables.

In order to model a network signature in AS, we decided to index all network packets in the network traffic window, so that the variables of the problem are indexes to the network packets instead of the network packets with all its individual values. This way, the variables of the problem will be a set of integer variables, each one representing a network packet.

Listing 2 represents a network CSP modelled in AS, where D represents the network traffic window, P the set of variables used to represent the network signature, where $\{P_1, \ldots, P_m\}$ are the network packet variables used to model the problem, and $\{P_{m+1}, \ldots, P_n\}$ the variables which are ignored. C is the set of the constraints used to model the network situation, and f_i the function that calculates the error of the constraint i, which takes as arguments the network traffic window, D, and the set of packets to which the constraint is applied, $\{P_j, \ldots, P_k\}$.

Listing 2 Representation of a network CSP modelled in AS

$D = \{(V_{(1,1)}, \ldots, V_{(1,19)}), \ldots, (V_{(n,1)}, \ldots, V_{(n,19)})\}$
$P = \{P_1, \ldots, P_m, \ldots, P_n\}$
$C = \{C_i, \ldots, C_l\}$;
$\forall C_i \in C \Rightarrow C_i = f_i(D, P_j, \ldots, P_k)$

In order to use only the N variables that describe the network situation, some of the variables in P have to be ignored. Several approaches have been used to do so: first, all the constraints used to describe the network situation will only be applied to the first N variables, the variables really necessary to model the problem. The second approach is to give a *null* error value to all variables that are not used to model the problem, the variables $\{P_{n+1}, \ldots, P_m\}$. Assigning a *null* error value to these variables will prevent AS from choosing them as the candidate for a value swap.

Modelling the problem using such a set of (indexing) variables and domains automatically solves the problem of restricting the domain of each packet to the network packet window. This way, maintaining the domain of each variable is facilitated, but in the other hand, the implementation of the constraints and its associated errors get more complex.

4.1.1 Modelling the constraints

The constraints used to model the problem need to access two types of data:

1 the set of variables used to model the CSP

2 the network traffic data.

The constraints are applied to the variables of the CSP, although, their associated costs are computed by checking the individual values of the real network packets which exists on the network traffic window. Each constraint applied to a set of network packet variables specifies a set of rules which must be verified in order for the constraint to be satisfied. These rules are checked by accessing the individual fields of the corresponding network variable packet.

The computation of the error of a single network packet variable is done by inspecting the individual values of the network packet which was assigned to such variable, and then, checking whether they violate the rules that compose the constraints which are applied to the network packet variable being analysed.

The cost of a candidate solution is the sum of the associated error of all variables that compose the problem, where the error of each variable is the sum of all constraint errors that are associated to that specific variable.

5 Modelling as a SAT problem

When encoding a network signature as a SAT problem, the first thing to be done is to create the SAT variables which compose the problem as well as the variables that represent the assignment of each possible value to a SAT variable.

To solve a SAT problem, MiniSat goes through two major steps:

1 read and parse the problem description represented in the CNF in order to build the internal representation of the problem in MiniSat

2 run the solve algorithms.

The problems are usually represented in a CNF clause file. Reading and parsing this file is very time consuming and has a great impact on the final performance of the MiniSat-based back-end.

We made some changes to MiniSat in order to eliminate the time spent in reading and parsing the description of the problem by adapting MiniSat so the CNF rules are generated inside MiniSat, thus avoiding this initial step.

5.1 Variables

Modelling a problem in SAT is quite different from doing so in other CP approaches. In SAT, the problems are encoded using only Boolean variables, i.e., variables that only can take values `True` or `False`. Because of this, we consider two types of variables:

1 the SAT variables which model the problem

2 the variables that represent the assignment of each possible value to a SAT variable.

In a network intrusion detection problem, each SAT variable represents a network packet. For each SAT variable, there is a variable for each network packet that can be assigned to each SAT variable. We decided that a SAT variable represents a network packet by indexing all packets on the network traffic instead of having a SAT variable to each network packet field, reducing significantly the complexity of the model.

When encoding a network signature as a SAT problem, the first thing to be done is to create the SAT variables which compose the problem as well as the variables that represent the assignment of each possible value to a SAT variable. For each network packet required to build the desired network signature, we create a SAT variable, and, for each SAT variable, it is created a variable for each possible value that can be assigned to it.

Listing 3 represent the variables used to model a network intrusion detection problem as a SAT problem, where D represents the set of network packets found on the network traffic, x the total amount of network packets found in the network traffic; DF_j the fields of packet j; $F(j, i)$ the field i of packet j; `SV` the set of all SAT variables, representing the network packet variables used to describe the desired network signature; n the number of SAT variables; and `V` the set of variables which represent the assignment of all possible values to each SAT variable SV and $V_i = \{V_{SV_iD_1}, \ldots, V_{SV_i,D_x}\}$ the variables that represent all possible assignments that SAT variable SV_i can take, i.e., V_{P_1,D_1} means that value D_1 was assigned to the SAT variable P_1.

Listing 3 SAT variables

$$D = \{D_1, D_2, ..., D_x\} \quad (1)$$

$$\forall D_j \in D \exists DF_j = \{F_{(j,1)}, F_{(j,2)}, ..., F_{(j,x)}\}, \quad (2)$$

$$SV = \{SV_1, SV_2, ..., SV_n\} \quad (3)$$

$$V = \{V_{SV_1,D_1}, ..., V_{SV_1,D_x}, V_{SV_2,D_1}, ..., V_{SV_2,D_x}, ..., V_{SV_n,D_1}, ..., V_{SV_n,D_x}\} \quad (4)$$

5.2 Variable domain

In a SAT problem, there is no concept of variable domain as in other constraint solving approaches. Since the variables used to model the problem are Boolean variables, their domain are the values `True` and `False`.

When representing network intrusion detection as a SAT problem, the constraints used to model the problem are also the ones responsible for ensuring that a valid solution makes sense on a given piece of network traffic, and also that the variables can only take values from the actual network traffic.

5.3 Constraints

Constraints are used to guide the encoding the problem as SAT, in a CNF form. The purpose of this encoding is to model the valid values according to the rules that should be verified by the constraint.

Encoding a problem as a SAT problem in a CNF form is quite complex and its size can grow very rapidly, due to the number of variables involved in a SAT problem. So, we created functions to model the necessary constraints, which according to some parameters create the necessary CNF clauses.

Two major types of encoding are necessary to model a problem in SAT:

1 encoding the set of variables which ensures the integrity of the solution

2 the ones that model the problem, encoding the constraints and modelling the desired intrusion signature.

The first step of the encoding is almost independent of both network traffic and the specific intrusion detection pattern which is to be modelled, depending only on the number of network packets in the network traffic window, and the number of network packets used to model the desired signature. This almost static step can be reused if those parameters are shared among several problems, allowing to obtaining major performance gains.

5.4 Variable encoding

Variable encoding is a very important step. Although we decided to use a set of *indexing* variables to represent the network packets, the variable encoding derives the domain of the variables by specifying a set of rules which states that each variable has to take *at least* one value, which is an index to a network packet; and each variable should take at *most* one index to a network packet, thus, stating the domain of each variable.

Encoding the set of variables is made in two steps:

1 ensuring that at least one value is assigned to each SAT variable by using the *at_least_one* clauses

2 ensuring that at most one value is assigned to each SAT variable through the use of *at_most_one* clauses.

The *at_least_one* clauses are a set of clauses represented in the *CNF* stating that each SAT variable should take at least one value from any network packet on the network traffic log. Listing 4 represents a formal representation of such clauses, where x represents the number of network packets in the network traffic and n the number of network packets used to model the intrusion signature.

Listing 4 *at_least_one* clauses – formal description

$$\bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{x} \left(V_{P_i,D_j} \right) \right)$$

The *at_most_one* clauses ensures that only one value is assigned to a SAT variable, which is accomplished by creating *conflict causes* between all combinations of possible values that can be assigned to a single variable, e.g., $\neg V_{(SV_1,D_1)} \vee V_{(SV_1,D_2)}$, means that if the value D_1 is assigned to variable SV_1, the network packet represented by D_2 cannot be assigned to the same variable SV_1. Listing 5 presents a formal representation of the *at_most_one* clauses, where x represents the number of network packets in the network traffic and n the number of network packets used to model intrusion signature.

Listing 5 *at_most_one* clauses – formal description

$$\bigwedge_{i=1}^{n} \left(\bigwedge_{j=1}^{x} \left(\bigwedge_{k=j+1}^{x} \left(\neg V_{P_i D_j} \vee \neg V_{P_i,D_k} \right) \right) \right)$$

5.5 Constraint encoding

The second major part of encoding a network signature as a SAT problem is the encoding of the problem itself, encoding the constraints that compose the network signature and which actually model the problem. The encoding of the constraints follows the same approach used to encode the variables, relying on *conflict causes* and *support clauses* to describe each constraint.

Due to the high complexity and size of the CNF rules, each constraint is modelled as a function, which in turn creates the necessary CNF rules required to encode the constraint. Using such functions, we can encode the desired network intrusion detection problem as a SAT problem, but hiding the complexity of writing CNF rules.

The constraints are encoded by analysing the network traffic, and based on that network traffic and the desired constraints, CNF rules are created by stating which network packets are compatible with each other, according to the constraint being encoded.

5.5.1 Modelling a problem

The modelling of the problem is achieved through the use of the functions which implements the constraints. First, the functions which encode and define the variables are applied; followed by the functions which implement the constraints and actually model the problem.

6 Sliding network traffic window

The GC, AS and MiniSat back-end detection mechanisms of NeMODe use a static network traffic log, which may be obtained with `tcpdump` (Jacobson et al., 1989), a network packet sniffer, while a computer is under an actual attack.

By using a static traffic sample, we are limiting the capabilities of the detection mechanism. This is necessary because watching live network traffic would be difficult to handle, performance-wise, and also because we need to establish benchmark results which require a fixed dataset.

Introducing a network traffic window that changes over time, which slides across a larger set gives the solver new capabilities, allowing it to analyse a much larger dataset than was previously possible. Besides, if we get the network traffic window to *slide* across live network traffic, it allows us to analyse live network traffic in real-time by updating the network traffic window with incoming network packets captured from the wire.

If, instead of simply slide the network window over a larger set, or updating it with fresh network packets, we *keep* in the network window past packets which seem interesting for the network situation that we are trying to detect, we get the capability of detecting attacks that spread across a window larger than that previously used, thereby including a range of packets that span a considerably larger time interval than was previously attainable.

6.1 Sliding network traffic window in AS

AS was chosen as the solver to implement the sliding network traffic window since, from the solvers we have experimented with, it is the one which is most easily modified and is less sensitive to changes, such as the changes on the network traffic window, due to the customisability of the AS algorithm.

AS relies on heuristics, reflected in the *error functions* in order to reach a solution to a combinatorial problem. In a network intrusion detection problem, these heuristics directly pertain to the network traffic window, since the *error functions* are calculated by analysing the packets actually found in the traffic window.

Due to this direct influence of the network traffic window over AS heuristics, any changes made to the network traffic window will have an immediate effect in the heuristic functions used by AS, changing the way it seeks for a solution and, most importantly, automatically adapting to any change made on the network traffic window.

AS reaches a solution to a problem by starting with an initial state, and then iteratively performing minor changes to it, until an objective function is satisfied. At each step, every variable of each 'tentative' solution is already assigned with a value, which is a reference to an actual network packet that belongs to some instance of the network packet window. So, when a network packet is removed from the packet window to make room for another packet, the 'tentative' solution is no longer valid. Due to the high performance and insensitiveness to previous context of the AS algorithm, it adapts very quickly to the new 'instance' of the network packet window, without requiring any changes to the code.

6.2 Updating the sliding window

In order to update the sliding window with new packets, we decided to use a *first in first out* access discipline, where the oldest packet in the network traffic window is replaced by the newest packet arriving in the network. Two versions of this approach were implemented, and, depending on the network case being analysed, the most suited version is used:

1 remove oldest packet, insert new packet

2 remove oldest not relevant packet, insert new packet.

6.2.1 Remove oldest packet, insert new packet

In a first version, when a new packet arrives, we simply remove the oldest network packet from the network traffic window and insert the new one in its position. At some point, while inserting new packets replacing the old ones, the packets in the network traffic window are no more ordered as in the original network traffic source, since we do not shift the packets when inserting a new one. This does not poses a problem to AS, since it uses each packet time stamp when there is the need impose temporal order between network packets.

6.2.2 Remove oldest not relevant packet, insert new packet

The second version of the sliding window was implemented in order to keep specific packets in the network packet window: the ones which are understood to be important for the desired network situation, even if they are among the oldest packets and would otherwise be replaced by new ones. This approach allows us to detect intrusions in a wider range than the network traffic window being used, since the relevant packets to such situation are being 'buffered' in the sliding network traffic window, allowing them to be related to newer packets which appear later in the network traffic.

6.2.3 Deciding if a packet is relevant

The decision of checking if a packet is relevant to the desired network situation is critical in keeping the *interesting* network packets in the traffic window, so as to relate them with packets that may appear in the future, beyond the limits of the network packet window size.

Deciding if a packet is relevant is directly related to the intrusion being described as well as its signature, since that decision is achieved through the use of subset of the heuristics that have been used in the description of the network attack as an AS problem. These heuristics are applied to the network packets being checked for relevance, and, depending on the result, the packet is considered relevant or not. These heuristics are usually very simple, checking specific packet fields such as ports, flags, addresses or time-stamp of a network packet.

6.3 Simulating live network traffic

The sliding network traffic window is implemented to use a `tcpdump` log file as network traffic source, but is designed to simulate live network traffic, up to a certain level, by simulating the network packet arrival at a given rate. This is accomplished by controlling when and which packet is considered a newly arrived packet, so it can be processed in order to be inserted in the network traffic window.

To simulate live network traffic using a `tcpdump` log file, we introduce a *sleep* time between the update of the network traffic window with new packets, thus, simulating the arrival of new network packets at a given network bandwidth.

This approach to simulate real live network traffic allows the fine tuning of the packet arrival rate, thus simulating different network traffic speeds, allowing to test NeMODe in situations similar to real network traffic, at different network bandwidths.

7 Examples

So far, we have worked with several network intrusion signatures, including:

1 SSH password brute-force

2 a distributed DNS spoof

3 a DHCP spoofing attack

4 a ARP poisoning attack.

All these intrusion patterns were described using NeMODe and the generated code was successful in finding the desired situations in the network traffic logs, using both static and sliding network traffic window.

7.1 Distributed SSH password brute-force

An SSH password brute-force attack happens when the attacker tries to access the SSH service of a given host by *brute-forcing* SSH username/password combinations, i.e., trying a large amount of username/password combinations, based on a dictionary or some other approach, to gain access to the SSH server.

7.1.1 Modelling the attack

This type of attack is characterised by a large number of SSH connection attempts. To detect this attack, we can monitor the number of SSH connections that are initiated and terminated in a small amount of time, which means the connection was not successful. If there are a few connections like these, it means that the host is probably under a SSH password brute-force attack.

We now present the modelling of a distributed SSH password brute-force in both NeMODe and Snort.

7.1.2 Modelling in NeMODe

Listing 6 shows how an SSH password brute-force attack can be described in NeMODe, we start by naming the network situation in line 1, and then specify the network traffic source, the file ‘ssh.pcap’, and the target solvers to which we will generate code.

Listing 6 An SSH password brute-force attack using NeMODe

```
1   ssh_brute_force {
2       RES = solve('ssh.tcpdump', [as,gecode,minisat]) {
3           P = {
4               tcp_packet(A),
5               dst_port(A)==22,
6               syn(A), nak(A)
7           },
8
9           C := clone(10,P),
10
11          same_dst(C:A),
12          max_duration(C) < secs(60)
13      }
14  } => {
15      alert('SSH password brute attack')
16  };
```

The network signature is actually described in lines 3 to 13. Line 15, alerts the network administrator for an eventual SSH password brute-force attack using the statement `alert` (‘`SSH password brute attack`’), if the specific attack is found.

Lines 3 to 7 describe a TCP packet `A` which initiates an SSH connection. These statements are assigned to variable *P*, which later, in line 9 we clone 10 times, meaning that we are looking for 10 packets, representing 10 SSH connection attempts.

In line 11, we state that packet `A` of each *instance* of clone `C` should all have the same destination address.

Then, in line 12, we state that the overall time of all clones should be less than 60 seconds, the value we found reasonable to consider it an attack, using the statement `max_duration`.

Finally, in line 15, we alert the network administrator for an eventual SSH password brute-force attack using the statement `alert` (‘`SSH password brute attack`’).

7.1.3 Modelling in Snort

It is possible to use Snort to describe and detect SSH password brute-force attacks by monitoring a large amount of SSH connections from the same source in a short period of time: this is achieved in a very limited way, resorting to built-in filters which impose a limit of network packets in given amount of time.

Listing 13 represents the rule which we used to detect this attack in Snort. It looks for packets going to port 22, where the SSH service is running, with the message ‘SSH-’ in it is payload. If there are five of these network connection from the same source in the interval of 60 seconds or less, then we could be under an SSH password bruteforce.

Listing 7 SSH password bruteforce Snort rule

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 22 \
(msg:"Possible SSH brute force attempt"; \
flow:to_server,established; \
threshold:type threshold, count 10, seconds 60; \
content:"SSH-"; offset: 0; depth: 18;)
```

7.2 Distributed DNS spoof

A DNS spoof is a man in the middle (MITM) attack, where the attacker tries to provide a false answer to a DNS query posted by the victim host. If the attack succeeds the victim could be accessing a host controlled by the attacker instead of the legitimate host. This allows the attacker to extract information from the victim.

We now present the modelling of a distributed DNS spoof in both NeMODe and Snort.

7.2.1 Modelling the attack

In order to perform this type of attacks, the attacker tries to respond with a false DNS answer faster than the legitimate DNS server, providing a false IP address for the name to which the victim was querying.

To detect this type of attacks, we want to look for several replies to the same DNS query, indicating the host might be under a DNS spoof attempt.

7.2.2 Modelling in NeMODe

Listing 8 shows how this attack can be modelled in NeMODe. We start by naming the intrusion in line 1, followed by the network traffic source in line 2. The actual description of the desired network situation is done in lines 2 to 14. Line 16 states what actions to take if the situation is found, in this situation the network administrators are alerted for an eventual DNS spoof attack.

Listing 8 A DNS spoof attack programmed in NeMODe

```
1   dns_spoofing {
2   RES = solve('dns.pcap', [as,gecode,minisat]) {
3   udp_packet(A), dst_port(A) == 53
4
5   udp_packet(B), src_port(B) == 53,
6   dst(B) == src(A), dst_port(B) == src_port(A),
7
8   udp_packet(C), src_port(C) == 53,
9   dst(C) == src(A), dst_port(C) == src_port(A),
10
11  B != C,
12  data(B,0,2) == data(A,0,2),
13  data(C,0,2) == data(A,0,2)
14  }
15  } => {
16  alert('DNS Spoofing attempt')
17  };
```

Line 3 describes the packet that makes the DNS request. Lines 5 to 6, models a first reply to the DNS request and lines 8 to 9 describes the second reply.

From line 11 to line 13, we state that the network packets `B` and `C` should be different and that the *DNS id* in replies should be the equal to the *DNS id* of the DNS request. The *DNS id* is represented in the first two bytes of the packet data.

7.2.3 Modelling in Snort

Snort provides some built-in rules which allows the detection of some DNS spoofing attacks, but they do so by analysing only specific properties in the headers and payload of the network packets, not being able to relate several packets to model the problem. More specifically, Snort ID 253 rule and Snort ID 254 rule, which checks for DNS replies with a TTL of 1 minute and no authority (Mathew et al., 2005), usual characteristics of a DNS spoofing attempt.

Listing 9 presents both Snort ID 253 and Snort ID 254 rules we used to detect the DNS spoof attack.

7.3 DHCP spoofing

A DHCP spoofing is another MITM attack, where the attacker tries to reply to a DHCP request faster than the legitimate DHCP server for the local network, allowing the attacker to provide false network configurations to the victim host, e.g., a fake default gateway, which forces all traffic from and to the victim host to pass through an attacker controlled host, allowing it to capture or modify sensitive data.

Listing 9 DNS spoof Snort rule

```
alert udp $EXTERNAL_NET 53 -> $HOME_NET any (msg:"DNS
SPOOF \
query response PTR with TTL of 1 min. and no authority";\
content:"|85 80 00 01 00 01 00 00 00 00|"; \
content:"|C0 0C 00 0C 00 01 00 00 00|<|00 0F|"; \
classtype:bad-unknown; sid:253; rev:4;)

alert udp $EXTERNAL_NET 53 -> $HOME_NET any (msg:"DNS
SPOOF \
query response with TTL of 1 min. and no authority"; \
content:"|81 80 00 01 00 01 00 00 00 00|"; \
content:"|C0 0C 00 01 00 01 00 00 00|<|00 04|"; \
classtype:bad-unknown; sid:254; rev:4;)
```

7.3.1 Modelling the attack

This kind of intrusion can be detected by looking for several answers to a single DHCP request, originating in different hosts. If the attacker spoofs its IP addresses, this detection method needs to be tuned (e.g., use MAC addresses).

7.3.2 Modelling in NeMODe

A NeMODe programme which models a DHCP spoof situation is presented in Listing 10. The signature is described in lines 2 to 8. Line 10 states which actions should be taken if the specific network situation is found.

Line 3 describes the packet that initiates a DHCP request, line 4 describes a first reply to such request and line 5 describes a second reply the DHCP request.

Finally, in line 7, states that packets `B` and `C`, the first and second replies, should have different source addresses.

Listing 10 A DHCP Spoofing attack programmed in NeMODe

```
1   Dhcp_spoofing {
2       RES = solve('dhcp.tcpdump', [as,gecode,minisat]) {
3           udp_packet(A), dst_port(A)==67,
4           udp_packet(B), dst_port(B)==68,
5           udp_packet(C), dst_port(C)==68,
6
7           src(B) != src(C)
8       }
9   } => {
10      alert('DHCP Spoofing attempt')
11  };
```

7.3.3 Modelling in Snort

For the DHCP spoofing attacks, Snort does not provide a ready to use preprocessor or rule. One of the only ways to detect a DHCP spoofing in Snort is to monitor for DHCP replies from hosts which are not a legitimate DHCP server (Noonan, 2004). This can actually detect some DHCP spoofing attacks, but can be easily evaded if the attacker also spoofs its IP address.

Listing 11 presents the rules necessary to detect this attack. First, we specify that we should accept all DHCP replies from the legitimate DHCP server with IP address `192.168.1.254`. Then we specify that any DHCP reply from other host will be considered a DHCP spoofing attempt.

To use this specific rule, Snort had to be run with the '–o' option, which reverses the order how the rules are read, considering 'pass' statements before 'alert' statements.

Listing 11 DHCP spoof Snort rule

```
pass udp 192.168.1.254 67 -> any 68
alert udp any 67 -> any 68 (msg: "Rogue DHCP server..."; sid:1)
```

This set of rules is effective to detect DHCP spoofing attempts, but they can fail if a new DHCP server is added to the network and the rule is not updated, leading to a large amount of false positives.

However, if an attacker decides to spoof its IP address, it can easily evade the detection.

7.4 ARP poisoning

An ARP poisoning attack happens when someone tries to poison the ARP tables of a router or specific host with fake data, making an IP address point to a MAC address corresponding to some other host which is not the legitimate *owner* of the given IP address.

This kind of attacks allows the attacker to gain unauthorised access to information, destined to someone else.

7.4.1 Modelling the attack

This type of attack is achieved by sending a series of ARP packets with fake information in order to poison the ARP tables of the desired hosts.

One way to detect ARP poisoning attacks is to monitor ARP packets, looking to see if there are different IP addresses assigned to the same MAC address in a short time. If this happens, the host is most likely under an ARP poisoning attack.

7.4.2 Modelling in NeMODe

Listing 12 presents a possible description of an ARP poisoning attack in NeMODe. It starts by naming the specific network situation in line 1, then stating what is network traffic source, and then specifying which solvers are going to be used.

The description of network signature is actually done in lines 2 to 21. In line 23, we alert the administrator if the specific attack is found.

Listing 12 An ARP poisoning attack programmed in NeMODe

```
1   arp_poisoning {
2       RES = solve('arp.tcpdump', [as,gecode,minisat]) {
3           arp_packet(A), arp_reply(A),
4           arp_packet(B), arp_reply(B),
5           arp_packet(C), arp_reply(C),
6           arp_packet(D), arp_reply(D),
7
8           time(A) < time(B),
9           time(B) < time(C),
10          time(C) < time(D),
11
12          src(A) == src(B),
13          src(A) == src(C),
14          src(A) == src(D),
15
16          src_mac(A) != src_mac(B),
17          src_mac(A) != src_mac(C),
18          src_mac(A) != src_mac(D),
19
20          time(D) – time(A) < secs(5)
21      }
22  } => {
23      alert('ARP poisoning attempt')
24  };
```

Lines 3 to 6 describe four packets which should be ARP replies, representing the ARP replies that we are looking for. Lines 8 to 10 state that these packets should be in that specific temporal order, so later we can specify a global time interval between the first and the last ARP reply, in line 20.

In lines 12 to 14, we state that packets `A`, `B`, `C` and `D` should all have the same sender protocol address (SPA), also known as IP Address, and in lines 16 to 18, we state that the source hardware address (SHA) of packet `A`, also known as MAC Address, must be different from the SHA of packets `B`, `C` and `D`. This means that we have packets `A`, `B`, `C` and `D` with the same IP address, but the MAC address of packet `A` is different from the MAC address of packets `B`, `C` and `D`, indicating that we are probably under an ARP poisoning attack.

To make this signature stronger, in line 20 we state that the time interval between network packets B and D should be less than 5 seconds, since an ARP poisoning attack tends to produces several ARP replies in a short time interval.

7.4.3 Modelling in Snort

Snort is also capable of detecting ARP poisoning attacks, but only if using the arpsoof preprocessor (Beale, 2004), which monitor for ARP packets against a user supplied ARP table containing valid (`MAC address`, `IP address`) pairs in the given network, which is hard to maintain when there are changes in the network.

Listing 13 presents the rules which were used to detect an ARP poisoning in Snort where it can be seen some known IP/MAC addresses combinations in the given network. If a different combination of the same IP/MAC addresses is found then we could be under an ARP poisoning attack. In this example, we present only 3 IP/MAC address combinations to simplify the example.

This Snort preprocessor is effective for detecting ARP spoofings, achieving a 100% detection rate (on our test runs). Still, it requires a large amount of maintenance if there are many hosts in the network to monitor, becoming unusable on large networks.

Listing 13 ARP poison Snort rule

```
preprocessor arpspoof
preprocessor arpspoof_detect_host: 192.168.1.70 \
                                        48:5d:60:72:d4:75
preprocessor arpspoof_detect_host: 192.168.1.90 \
                                        08:00:27:94:2c:46
preprocessor arpspoof_detect_host: 192.168.1.254 \
                                        00:24:17:70:26:EC
```

8 Experimental results

We have experimented with several network situations, including the ones described in Section 7: a SSH password brute-force, a DNS spoof, DHCP spoof, and an ARP poisoning attack. All these intrusions were successfully described using NeMODe. Based on this description, it was produced valid code for GC, AS and MiniSat, which was then executed in order to validate the code and ensure that it could indeed find the desired network intrusions. We used both static and sliding network traffic window in all network situations.

The code was run on a dedicated computer, an HP Proliant DL380 G4 with two Intel(R) Xeon(TM) CPU 3.40 GHz with 4 GB of memory, running Debian GNU/Linux 4.0 with Linux kernel version 2.6.18-5.

For the SSH password brute-force, we created two log files, composed of 400 and 182 TCP network packets while a computer was being under an actual attack. These log files were then used as the network traffic source to detect the attack. As for the DNS spoof, the DHCP spoof and the ARP poisoning, we created two log files of 400 and 100 TCP network packets while a computer was under a these specific attacks. These files were then used as the traffic source.

8.1 Static network traffic window

Table 1 presents the time (user time in milliseconds) required to find the desired network situations for the attacks presented in this work, using GC and AS).

Table 1 Average time (in ms) necessary to detect the intrusions using GC and adaptive search

Intrusion	*Log size*	*Case size*	*GC (ms)*	*AS (ms)*	*Detection rate (%)*
SSH password	400	10	18.75	4.37	100
	182	10	12.6	1.49	100
DNS spoof	400	3	6.94	5.78	100
	100	3	4.37	2.26	100
DHCP spoof	400	3	8.26	1.09	100
	100	3	3.98	0.46	100
ARP spoof	400	4	23.125	18.04	100
	100	4	5.46	2.34	100

Table 2 Average time (in ms) necessary to detect the intrusions using MiniSat

Intrusion	*Log size*	*Case size*	*Setup (ms)*[1]	*Solve (ms)*[2]	*Total (ms)*[3]	*Detection rate (%)*
SSH password	400	10	332.5	243.12	575.62	100
	182	10	51.25	23.12	74.37	100
NS spoof	400	3	106.32	28.07	134.39	100
	100	3	6.95	0.70	7.65	100
DHCP spoof	400	3	105.93	11.17	117.10	100
	100	3	6.79	0.48	7.27	100
ARP spoof	400	4	195.93	34.53	230.46	100
	100	4	11.71	2.89	14.60	100

Notes: [1]MiniSat setup time; [2]MiniSat solve time; [3]SAT total time (Setup + Solve)

Table 2 presents the same results while using MiniSat, where `Setup` is the time required to *encode* the problem as a SAT problem, solve is the time spent to solve the problem, and `Total` is the total time used by MiniSat.

In both tables, `Log size` is the size of the network traffic log, and the `Case size` is the number of network packets described by the specific signature, which is also the number of packets of a solution for the problem. The times presented are the average of 128 runs.

8.2 Sliding network traffic window

The sliding network traffic window was only implemented in the ×86 version of adaptive search. All results presented in this section are related to the use of adaptive search with a sliding network traffic window.

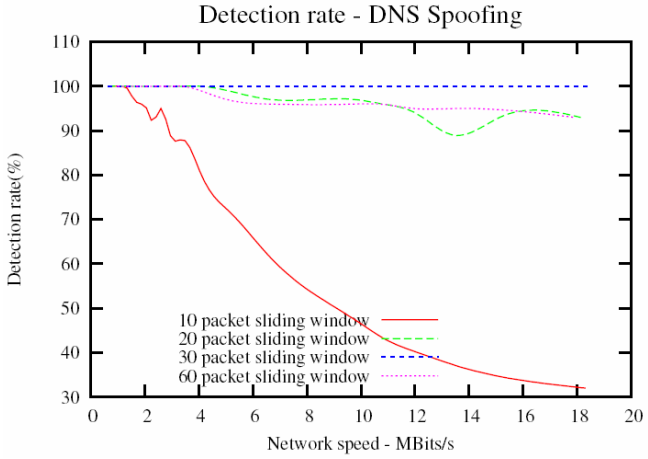
The results of using a sliding network traffic window in NeMODe are presented in terms of detection rate, instead of the time necessary to detect the first occurrence of the desired network situation.

For a better understanding of the results, we present them as charts rather than in tables.

8.2.1 DNS spoof

Figure 2 presents the detection rate of a DNS spoofing attack, while simulating several network speeds, measured in Mbit/s. The chart presents the results using a sliding window of 10, 20, 30, and 60 network packets, over a network traffic log of 400 network packets. The results presented are the average of a 100 runs.

Figure 2 DNS spoof – detection rate (see online version for colours)

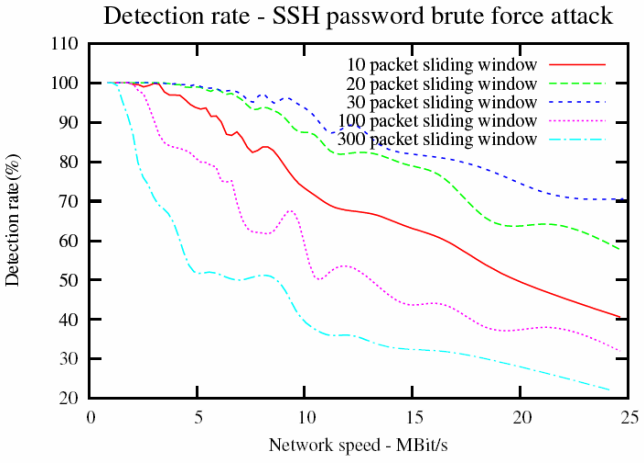


8.2.2 SSH password brute-force

Figure 3 represents the detection rate of the SSH password brute-force attack, while simulating several network speeds, measured in Mbit/s.

We present the results of using a sliding window of 10, 20, 30, 100 and 300 network packets, over a network traffic log of 3,000 network packets. The results presented are the average detection rate of 100 runs.

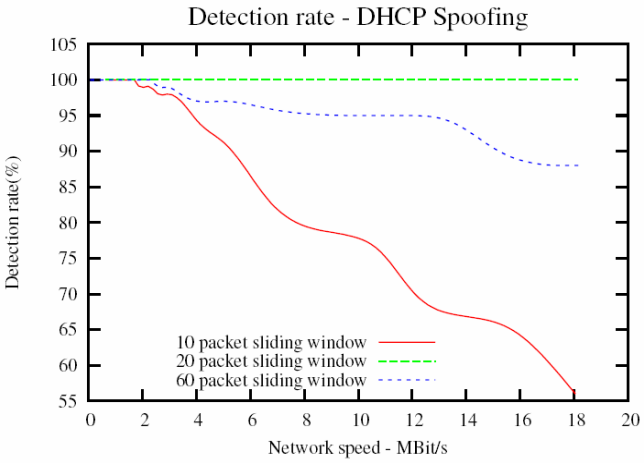
Figure 3 SSH password brute-force – detection rate (see online version for colours)



8.2.3 *DHCP spoof*

Figure 4 represents the detection rate of a DHCP spoofing attack, while simulating several network speeds, measured in Mbit/s. We present the results for a sliding window of 10, 20, and 60 network packets, over a network traffic log of 400 network packets. The results presented are the average detection rate of a total of 100 runs.

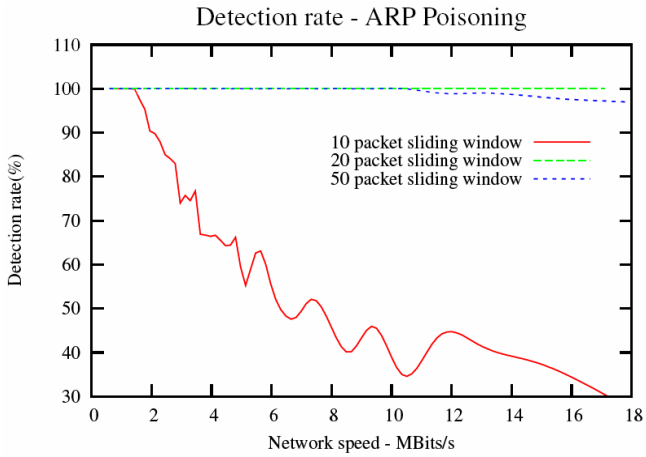
Figure 4 DHCP spoof – detection rate (see online version for colours)



8.2.4 *ARP poisoning*

Figure 5 represents the detection rate of the ARP poisoning attack, while simulating several network speeds, measured in Mbit/s. We present the average results using a sliding window of 10, 20 and 50 network packets, over a network traffic window of 500 network packets. The results presented are the average of 100 runs.

Figure 5 ARP poisoning – detection rate (see online version for colours)



8.3 *Using Snort*

To evaluate these network situations in Snort, we used Snort in *replay mode*, enabling the use of *tcpdump* or *pcap* files, allowing the use of the same network traffic log files used in NeMODe.

Table 3 presents the results obtained by Snort in microseconds. We present the time spent by the specific rule, by the pre-processors and the total time needed to process the network traffic log and detect the specific attack. We also present the detection rate of Snort. We present the results for each network situation with log files of different sizes, the same used to evaluate NeMODe.

In all experiments with Snort, we removed all unnecessary rules and preprocessors which were not relevant for the specific attack being detected.

Table 3 Snort results (in µs)

Signature	*Log size*	*Specific rule (µs)*	*Pre-processor (µs)*	*Total time (µs)*	*Detection rate (%)*
SSH password	400	24	1425	1449	100
	182	13	619	632	100
DNS spoof	400	n.a.	1042	1042	0
	100	n.a.	329	329	0
DHCP spoof	400	34	1020	1054	100
	100	24	354	378	100
ARP spoof	400	2	1018	1020	100
	100	1	353	354	100

9 Evaluation

The experimental results described in Section 8 shows that the performance varies in a great scale depending on the problem and on the recogniser, showing that each detection mechanism is very sensitive to the size of the network traffic log, size of the problem, and on the problem itself.

Tables 1 and 2 shows that the approach based on AS performs better than GC and MiniSat, which is explained by the finely tuned heuristics used to model each situation in AS.

The recogniser based on MiniSat is the one which is more affected by number of packets in the variable domain, mostly due to the exponential increase in the number of rules used to model the problem when this number rises. It is also possible to conclude that most of the time consumed by MiniSat is used to setup the problem. Although this is an essential part to reach a solution, it can be minimised by *pre-calculating* some setup clauses which are almost independent of the problem. These clauses depend only on the size of the network traffic window and on the number of variables needed to model the problem.

As for the sliding network traffic window, it presents very promising results for all case studies. We manage to reach a detection rate of 100% with network speeds of 5 MBit/s for the SSH brute-force password attack and 18 MBit/s for the other attacks, which are very good results for a preliminary prototype, with the single purpose of proving the concept. Although in a very early stage, these results give us a high confidence to start working with real-time networks, instead of using network logs or simulated network traffic.

9.1 NeMODe vs. Snort

The attacks presented in this work can be detected by tools which use different approaches to network intrusion detection. These tools usually present limited ways to describe the desired network situation, if possible to describe them at all.

Also, they usually cannot describe or detect attacks that spread across several network packets, and when they do, the description of such attacks is very limited, resorting to preprocessors built with the single purpose of detecting a specific network situation.

Although the cases we have experimented in this work can be detected by systems such as Snort, they cannot be modelled in a descriptive way as in NeMODe. This difference in modelling methodologies makes a direct comparison difficult, nevertheless, we decided to experiment on the same network situations in Snort using the same network traffic logs in order to obtain some experimental results.

Looking to the results obtained by Snort in Table 3, one can easily conclude that Snort performs better than NeMODe in terms of time required to process the network traffic and detect the specific intrusion. Still, if we compare the description of the same attacks in the two systems, which can be found in Section 7, we can easily conclude that NeMODe is much more expressive, allowing to easily state relations between several network packets.

Although the rule presented in Listing 13 for the SSH password brute-force is effective in some cases, it does not make use of real relations between several network packets, making the description counter-intuitive and hard to express. In our experiments, we obtained a detection rate of 100%, mostly because the rule was specifically tailored to match the attacks in the network traffic logs being analysed. If the attack is done in a slightly different way, the signature can fail to detect the attack.

The Snort rules set presented previously in Listing 9 for the DNS spoof were used ‘as is’ in the Snort rule data-base, having the specific purpose of detecting DNS spoof attempts. In fact, these rules can detect some DNS spoofing attempts, but in many cases, depending on the tools used to perform the attack, they will fail. In particular, this set of rules were not able to detect any of our DNS spoofing attempts made with the help of `ettercap`.

In the DHCP spoof attack, the way to describe the network situation in Snort is very effective, but it requires on specifying the legitimate DHCP servers. This forces to rewrite the rule if the DHCP servers change their addresses.

As for the ARP spoofing, Snort provides very efficient preprocessors, but they require a lot of maintenance, becoming unusable in large scale networks.

10 Conclusions and future work

In this work, we introduce a system for network intrusion detection based on CP, allowing an intuitive and expressive way to describe such situation due to the possibility of the specification of relations between several network packets in an easy way.

We demonstrate how to easily describe network situations in NeMODe, including network attacks which span across several network packets, using a declarative approach. From that single description, it generates several network situation recognisers based on CP, using different CP paradigms, which are then used to detect the specific intrusions.

This work also shows that NeMODe can easily be adapted to perform intrusion detection on live networks, by using a sliding network traffic window, which is constantly updated with freshly arrived network packets.

Although Snort is able to describe and detect the network situations analysed in this work, they are much harder to describe and have to resort to built-in filters, making such description much more complex.

The results obtained in this work are very promising, providing a platform to start performing network intrusion detection on a live network traffic link in a near future, a very important future step.

We still need to model more network situations as a CSP to better evaluate the performance of the system. Also, we have plans to implement new back-end detection mechanisms using different CP paradigms.

References

Arun, K. (2009) 'Flow-aware cross packet inspection using bloom filters for high speed data-path content matching', in Advance *Computing Conference, 2009. IACC 2009. IEEE International*, pp.1230–1234.

Beale, J. (2004) *Snort 2.1 Intrusion Detection*, 2nd ed., Syngress Publishing, Rockland, MA, USA.

Biere, A. and Press, I. (2009) *Handbook of Satisfiability*, IOS Press, Amsterdam, The Netherlands.

Codognet, P. and Diaz, D. (2001) 'Yet another local search method for constraint solving', *Lecture Notes in Computer Science*, Vol. 2264, pp.73–90, Springer.

Comer, D. (2006) *Internetworking with TCP/IP Volume 1: Principles Protocols, and Architecture*, 5th ed., Prentice Hall, Upper Saddle River, New Jersey, USA.

Een, N. and Sörensson, N. (2006) 'Minisat v2. 0 (beta)', *Solver Description, SAT Race*, Vol. 2006.

Jacobson, V., Leres, C. and McCanne, S. (1989) *The tcpdump Manual Page*, Lawrence Berkeley Laboratory, Berkeley, CA.

Kumar, S. and Spafford, E. (1995) 'A software architecture to support misuse intrusion detection', in *Proceedings of the 18th National Information Security Conference*, pp.194–204.

Mathew, S., Britt, D., Giomundo, R., Upadhyaya, S., Sudit, M. and Stotz, A. (2005) 'Real-time multistage attack awareness through enhanced intrusion alert clustering', in *Military Communications Conference, 2005. MILCOM 2005. IEEE*, IEEE, pp.1801–1806.

Noonan, W. (2004) *Hardening Network Infrastructure*, McGraw-Hill Osborne Media, Emeryville, California, USA.

Roesch, M. (1999) 'Snort – lightweight intrusion detection for networks', in *LISA '99: Proceedings of the 13th USENIX conference on System administration*, USENIX Association, Berkeley, CA, USA, pp.229–238.

Rossi, F., Van Beek, P. and Walsh, T. (2006) *Handbook of Constraint Programming*, Elsevier Science, Amsterdam, The Netherlands.

Salgueiro, P., Diaz, D., Brito, I. and Abreu, S. (2011) 'Using constraints for intrusion detection: the NeMODe system', in Rocha, R. and Launchbury, J. (Eds.): *PADL'11 – Thirteenth International Symposium on Practical Aspects of Declarative Languages, Lecture Notes in Computer Science*, Vol. 6539, Springer-Verlag, Berlin, Heidelberg.

Salgueiro, P.D. and Abreu, S.P. (2010) 'A DSL for intrusion detection based on constraint programming', in *Proceedings of the 3rd International Conference on Security of Information and Networks, SIN '10, ACM*, New York, NY, USA, pp.224–232 [online] http://doi.acm.org/10.1145/1854099.1854145 (accessed August 2011).

Schulte, C. and Stuckey, P. (2004) 'Speeding up constraint propagation', *Lecture Notes in Computer Science*, Vol. 3258, pp.619–633, Springer.

Sörensson, N. and Een, N. (2005) 'Minisat v1. 13-a sat solver with conflict-clause minimization', *SAT*, Vol. 2005, p.53.

Van Hentenryck, P. and Michel, L. (2005) *Constraint-based Local Search*, MIT Press, Cambridge, MA, USA.

Zhang, Y. and Lee, W. (2000) 'Intrusion detection in wireless ad-hoc networks', in *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, ACM, p.283.